

Zvyšování výpočetního výkonu mikroprocesoru

ZVYŠOVÁNÍ VÝPOČETNÍHO VÝKONU PROCESORŮ

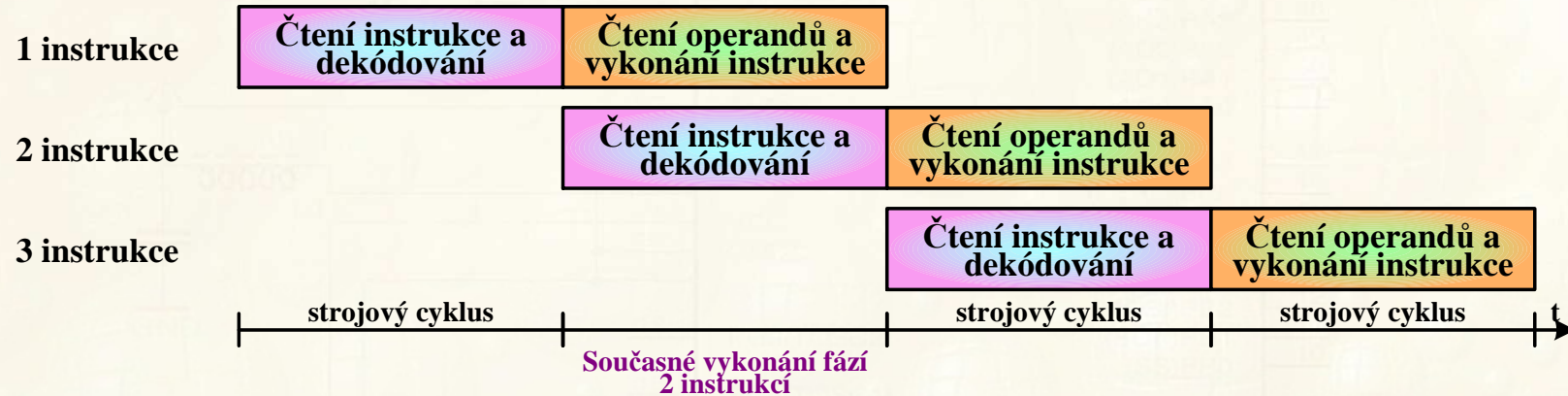
- ❖ **Oblast technologie** (maximalizace hodinového kmitočtu). Předpokládané odhady (bipolární Si - 800MHz, CMOS – 3,3GHz).
- ❖ **Oblast architektury procesoru** - paralelizace procesů a zpracování jednotlivých instrukcí.
 - **Překrývání instrukcí** „pipeline“ (segmentace zpracování instrukcí)
 - **Zvětšování počtu** paralelně pracujících **jednotek superskalární** architektury – řazení instrukcí zajišťuje obvodově řadič procesoru (např. PC).
 - **Paralelizace instrukcí** (architektura **VLIW**) – řazení současně zpracovávaných instrukcí zajišťuje programátor nebo překladač (signálové procesory).
 - **Paralelně řazené** segmentované aritmetické jednotky tzv. superscalar super-pipelining (Alpha, P4).
 - **Paralelizace** založená na zvětšování **počtu jader** (multi-core).
- ❖ **Paralelizace uplatňovaná na zpracovávaná data nebo instrukce.**

Segmentace instrukce do překrývajících se fází byla přirozeným vyústěním snahy o zvyšování výpočetního výkonu. Zahájen u 8051 (1980) částečným překrýváním instrukcím.

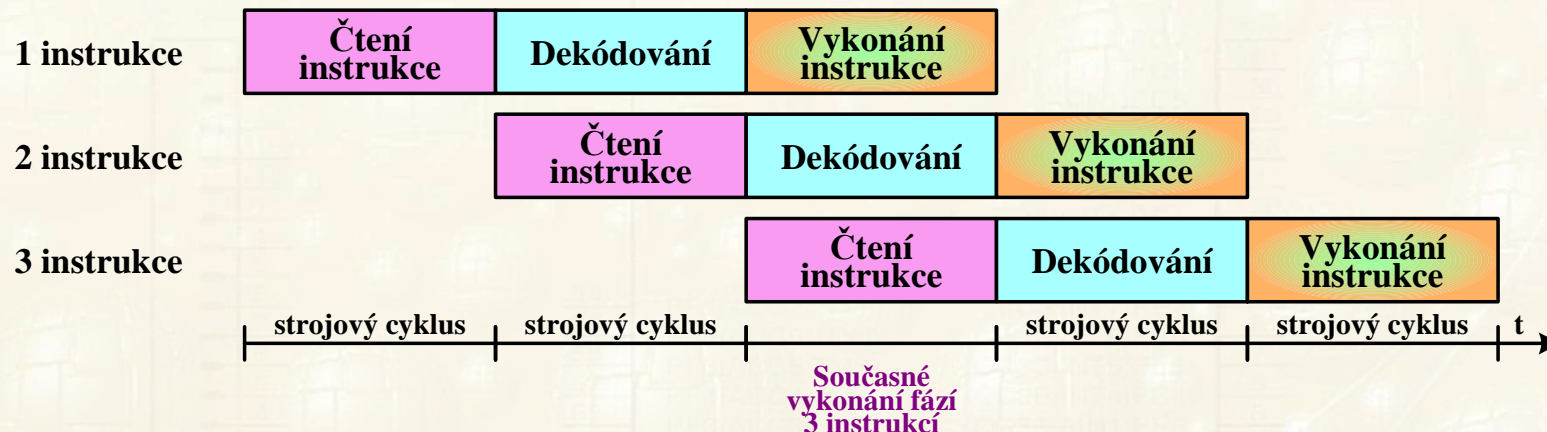
- Segmentovaná aritmetická jednotka nebo celý procesor má instrukce rozděleny na dvě nebo více částí.
- Výsledky fáze se ukládají do vyrovnávací paměti, z které si je v dalším strojovém cyklu přebírá ke zpracování další segment.
- Jsou-li **všechny segmenty naplněny**, pak každý strojový cyklus je **jedna instrukce dokončena**.
- Konfigurace se označuje „**pipeline**“ nebo „Princip sdílení času“.
- „superpipeline“ počet fází >4 . Pro počet fází > 3 se komplikuje obsluha operační paměti a větvení programu.
- **U jednočipových (8-bitových) μ P maximálně 2 fáze.**
- Je-li v procesoru více stupňový „pipeline“ \Rightarrow co s načtenými instrukcemi, které nemají být vykonány (podmíněné větvení a návrat, nepodmíněný skok, atd.) \Rightarrow instrukce **NOP**, **zpožděné instrukce**, **podmíněně vykonávané instrukce**

ZVYŠOVÁNÍ VÝPOČETNÍHO VÝKONU PROCESORŮ

- Procesory AVR využívají 2 fázový pipeline. Čtení instrukce a její dekódování ⇒ Čtení operandů a vykonání instrukce

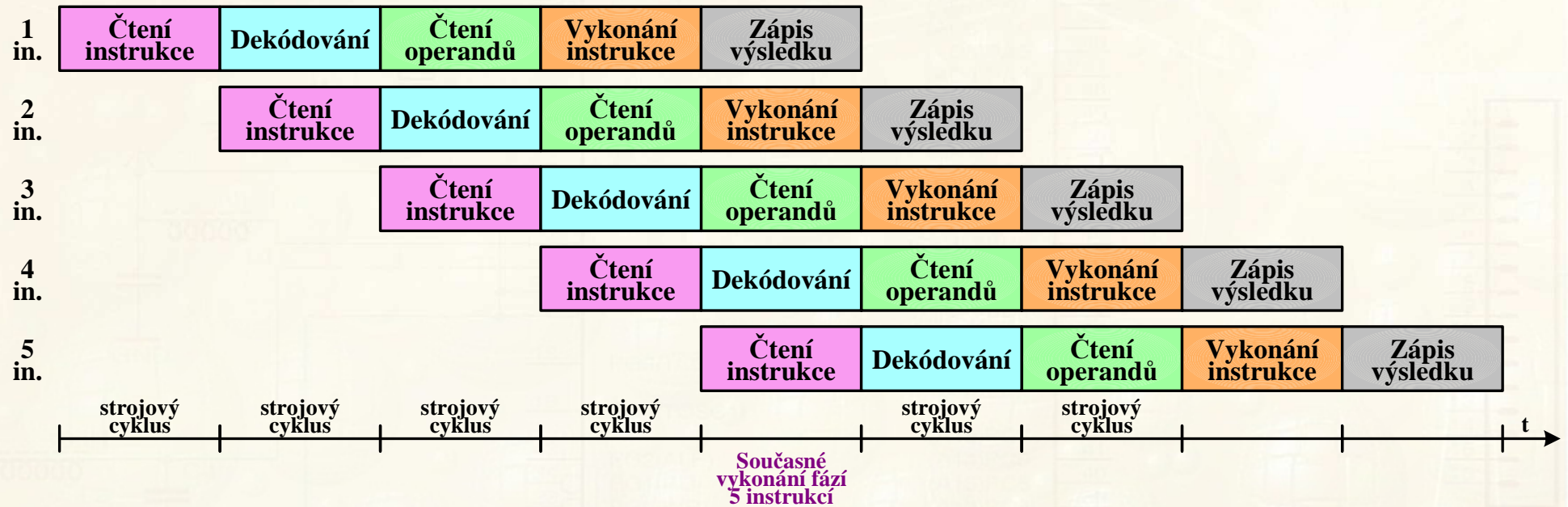


- Procesory Cortex-M3 používají 3-fázový pipeline pro vykonání instrukcí. Čtení instrukce (Fetch) ⇒ Dekódování instrukce ⇒ Čtení operandů a vykonání



ZVYŠOVÁNÍ VÝPOČETNÍHO VÝKONU PROCESORŮ

- Procesory ARM9 využívají 5 fázový pipeline



- U běžných signálových procesorů se pipeline pohybuje v rozmezí 3÷6
- U signálových procesorů (VLIW) s instrukčním cyklem (0,87ns až 3ns) jsou časově kritické přístupy i do paměťových prostorů, které se rozkládají do dalších fází.
Např. TMS320C6416 (příprava adresy ⇒ přenos adresy po sběrnici ⇒ vybavení obsahu paměti ⇒ přenos dat po sběrnici). Díky tomu se pipeline prodlužuje na 7 fázový.
Výsledky některých vykonaných instrukcí nejsou ihned k dispozici ⇒ pipeline je skrytě 11 fázový.

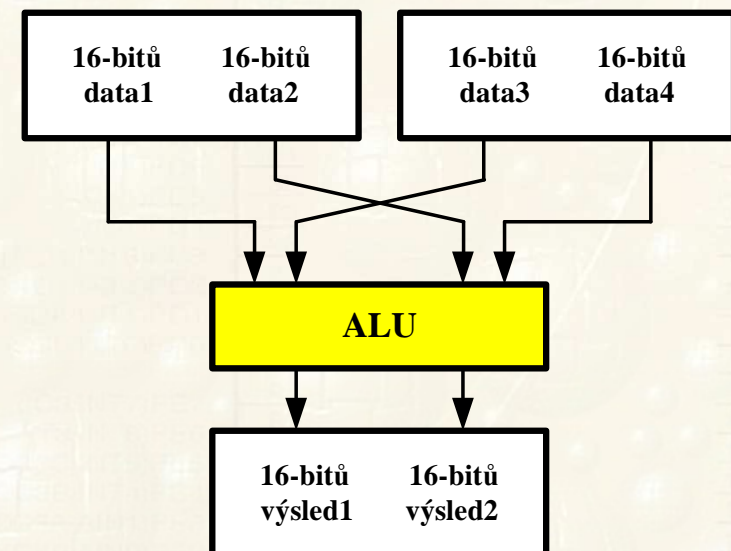
ZVYŠOVÁNÍ VÝPOČETNÍHO VÝKONU PROCESORŮ

SIMD (Single Instruction stream, Multiple Data stream) je podpůrný prostředek využívaný od 90 let. m.s. ke zvýšení výpočetního výkonu procesorů MMX (PC) a u signálových procesorů. SIMD je systém zpracovávající několik datových toků jedním programem.

- Operandy **jedné instrukce** vzniknou sloučením nezávislých dat do jednoho delšího datového slova
- Výkonná jednotka je rozdělena na dvě nebo více částí.
- Jedna instrukce provede stejnou operaci (např. sčítání, násobení) se dvěma nebo více nezávislými hodnotami.
- **Efektivnost skládání slov ?**
- Využití např. Viterbiho algoritmus, Zpracování radarových dat

$$\text{výsled1} = \text{data1} + \text{data3}$$

$$\text{výsled2} = \text{data2} + \text{data4}$$

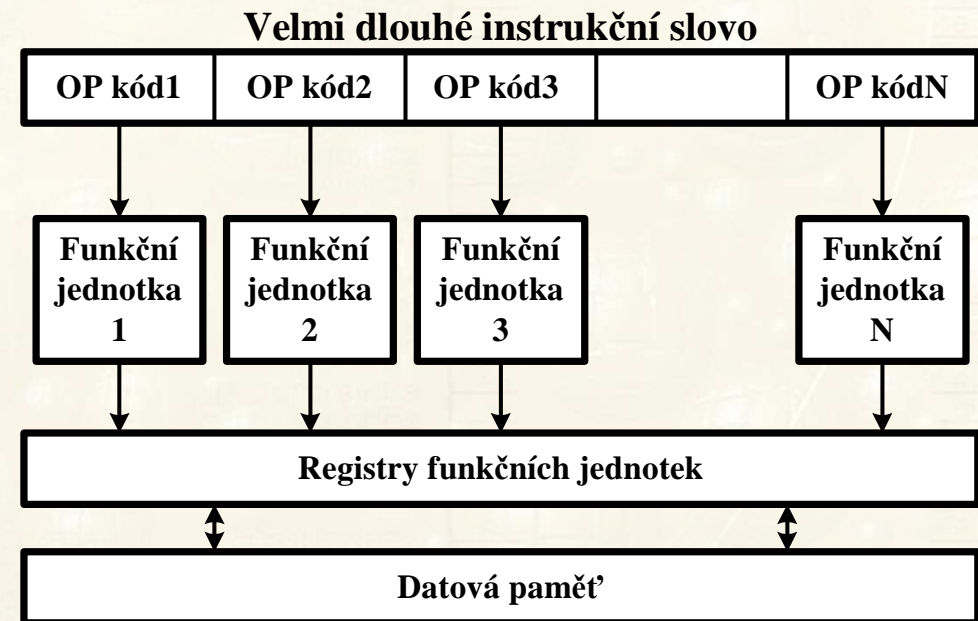


ZVYŠOVÁNÍ VÝPOČETNÍHO VÝKONU PROCESORŮ

MIMD (Multiple Instruction stream, Multiple Data stream)

Obecný typ paralelního systému zpracovávajícího několik datových toků samostatnými jednotkami, z nichž každá realizuje samostatný program.

- Standardní multiprocessorové systémy
 - Složené z **několika procesorů**
 - Integrované do čipu (pole procesorů TMS320C8x, AD14060, atd.).
- **VLIW (Very Large Instruction Word)** a jeho modifikace. V procesoru je integrováno více **aritmetických jednotek** schopných zpracovat samostatně instrukci.



Signálové procesory

SIGNÁLOVÉ PROCESORY

- ❖ 1979 – první signálový procesor I2920 (Intel) \Rightarrow malé kapacity pamětí, integrovaný A/D a D/A převodník, bez násobičky a možnosti připojení externích pamětí. Neujal se.
- ❖ Nový možný směr vývoje, na který navázaly firmy TI, Analog Device, NEC.
- ❖ Nyní používány pro číslicové zpracování signálu a obrazu, tak i pro obecné logické řízení a regulaci.
- ❖ **Architektura** a instrukční soubor optimalizován na rychlou realizaci **násobení, sčítání (akumulace) a posunu**

Konvoluce
$$y_n = \frac{1}{N} \sum_{i=0}^{N-1} x_i \cdot x_{n+i}$$

Filtrace
$$y_n = \sum_{i=0}^M a_i \cdot x_{n-i} - \sum_{i=1}^L b_i \cdot y_{n-i}$$

Diskrétní transformace
$$X_k = \sum_{i=0}^{M-1} x_i \cdot W^{ik}$$

Korelace
$$y_n = \sum_{i=0}^{M-1} h_i \cdot x_{n-i}$$

$$y_n = \sum_{i=0}^{M-1} h_i \cdot x_{n-i}$$

Realizace výpočtu na běžném procesoru

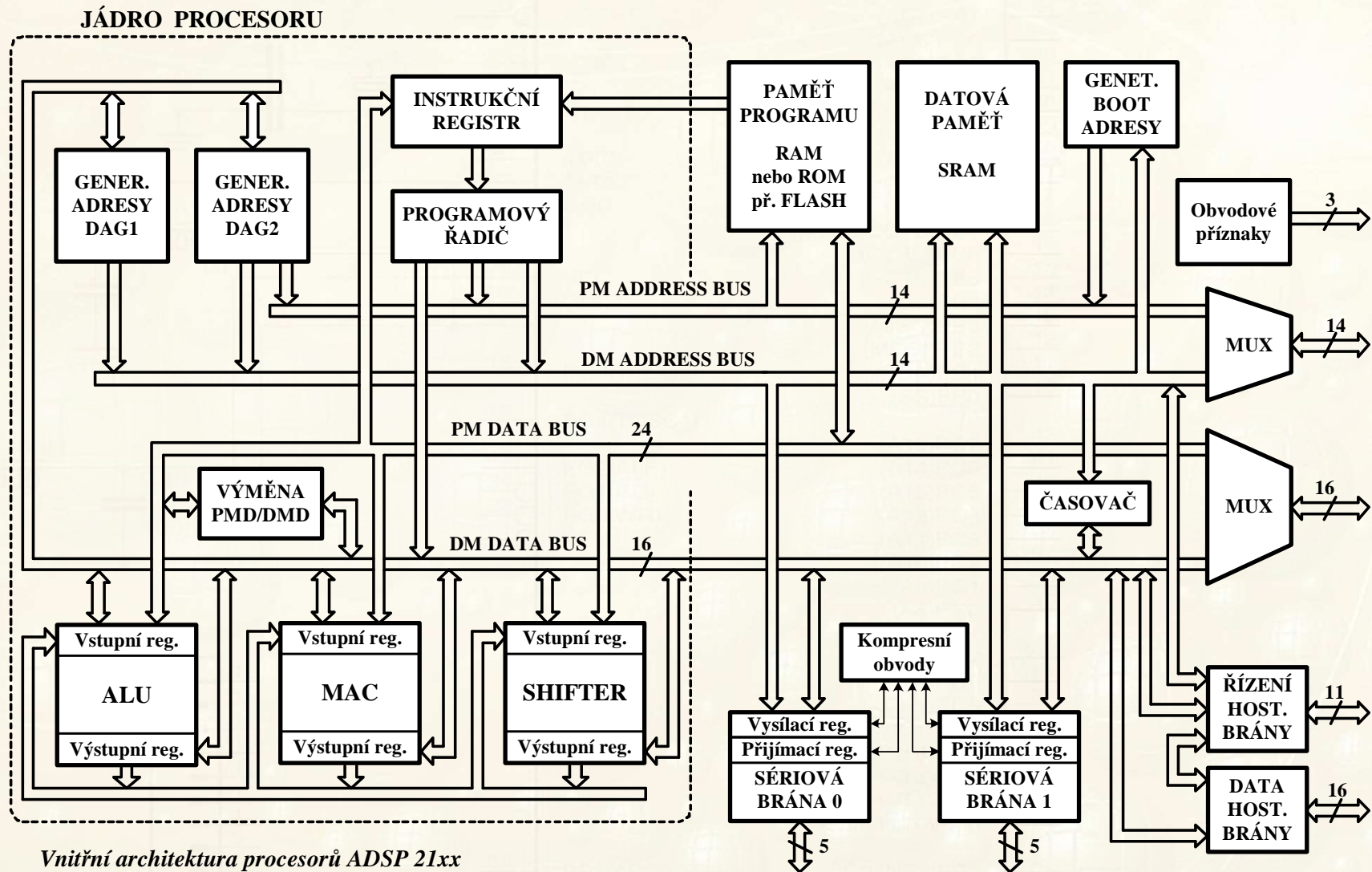
opak aC=Adr h₀ , aD=Adr x₀ , počítadlo=M , Y=0
 C=(aC); D=(aD); ; Vzorek x_n a h_i do registrů násobičky
 Z= C*D; ; Součin x_n a h_i
 Y=Y+Z; ; Přičtení součinu (více bitů a instrukcí)
 aC++; aD++; ; Adresy na novou konstantu a data
 M=M-1; ; Zmenšení počítadla
 if (M!=0) goto opak ; výpočet M součinů

Realizace výpočtu na procesoru ARM

opakvys LDRH R4,[R1],#2 ; Vzorek xn do R4, R1=R1+2 (16bitů)
 LDR R5,[R2],#4 ; Koeficient do R5, R1=R1+4 (32bitů)
 MLA R3, R4, R5, R3 ; Součin a přičtení k R3
 SUB R6, #1 ; Zmenšení počítadla
 CBZ R6, opakvys

- ❖ **Realizace** součinu, součtu, adresování nových operandů a počítačla opakování v jednom strojovém cyklu.
- ❖ **Modifikovaná** harvardská struktura s odděleným programovým a datovým prostorem.
- ❖ **Rozšířený střadač** = počet bitů součinu operandů + 8 bitů
- ❖ **Současná manipulace** se dvěma nebo více operandy.
- ❖ Výpočetní výkon zvyšován technologií a zřetězením instrukcí **„pipeline“** princip „sdílení času“ (dnes 2 až 7 (skrytě 12)).
- ❖ V současné době opět nejvýkonnější procesory, ačkoliv pracují na kmitočtu 3x nižším, než procesory PC.

ARCHITEKTURA BĚŽNÉHO SIGNÁLOVÉHO PROCESORU



Adresovací módy generátorů adres signálových procesorů str 445 až 450

FILTR FIR - ALGEBRAICKÝ ASSEMBLER, KRUHOVÉ ADRESOVÁNÍ

I0=data;	{ Nastavení ukazatele na data}
L0=LENGTH(data);	{ Nastavení délky zásobníku}
M0=1; M1=0; M2=-1; M4=1;	{ Nastavení modifikačních registrů}
I4=coef; L4= LENGTH(coef);	{ Nastavení ukazatele a délky zásobníku}
CNTR=taps-1;	{ Nastavení počtu opakování cyklu}
MR=0, MX0=DM(I0,M0), MY0=PM(I4,M4);	{ Nulování střadače MR, načtení vzorku dat a konstanty}
DO sop UNTIL CE;	{ Spuštění cyklu po návešti sop}
sop: MR=MR+MX0*MY0(SS), MX0=DM(I0,M0), MY0=PM(I4,M4);	
{ Násobení (se znaménky), sčítání a načtení nových dat a konstanty}	
MR=MR+MX0*MY0(RND);	{ Poslední součin, součet a zaokrouhlení}
IF MV SAT MR;	{ Test overflow a případná saturace}
dm(ad_out)=MR1;	{ Uložení bitů 31÷16 MAC do D/A}
AX0=dm(ad_in);	{ Uložení nového vzorku vstup. signálu}
dm(I0,M0)=AX0;	{ Uložení nového vzorku do paměti a přesun ukazatele na nejstarší vzorek x(n-M+2)}

SIGNÁLOVÉ PROCESORY – FILTRACE _ ASSEMBLER

STM #COEF+4, AR3 ; Nastavení ukazatele AR4 na koeficient B(M-1)
STM # X+49, AR2 ; Nastavení ukazatele dat na nejstarší vzorek
STM # -1, AR0 ; Ukazatele se budou zmenšovat o hodnotu 1
STM # 50, BK ; Nastavení délky kruhových zásobníků
SSBX FRCT ; Zpracovávané hodnoty jsou ve formátu Fraction
; Součin bude posunut o jednu pozici doleva

FIR:

RPTZ A, #49 ; Čítač opakování nastaven na 50 cyklů
MAC *AR2+0%,*AR3+0%,A; Násobení a sčítání s kruhovým adresováním
STH A,*AR2 ; Uložení výsledku na pozici nejstaršího vzorku
PORTW *AR2, PA0 ; Zápis výsledku do D/A převodníku
BD FIR ; Zpožděný skok na začátek programu FIR
PORTR PA1,*AR2+0% ; Uložení nového vzorku a nastavení ukazatele
; na nejstarší vzorek

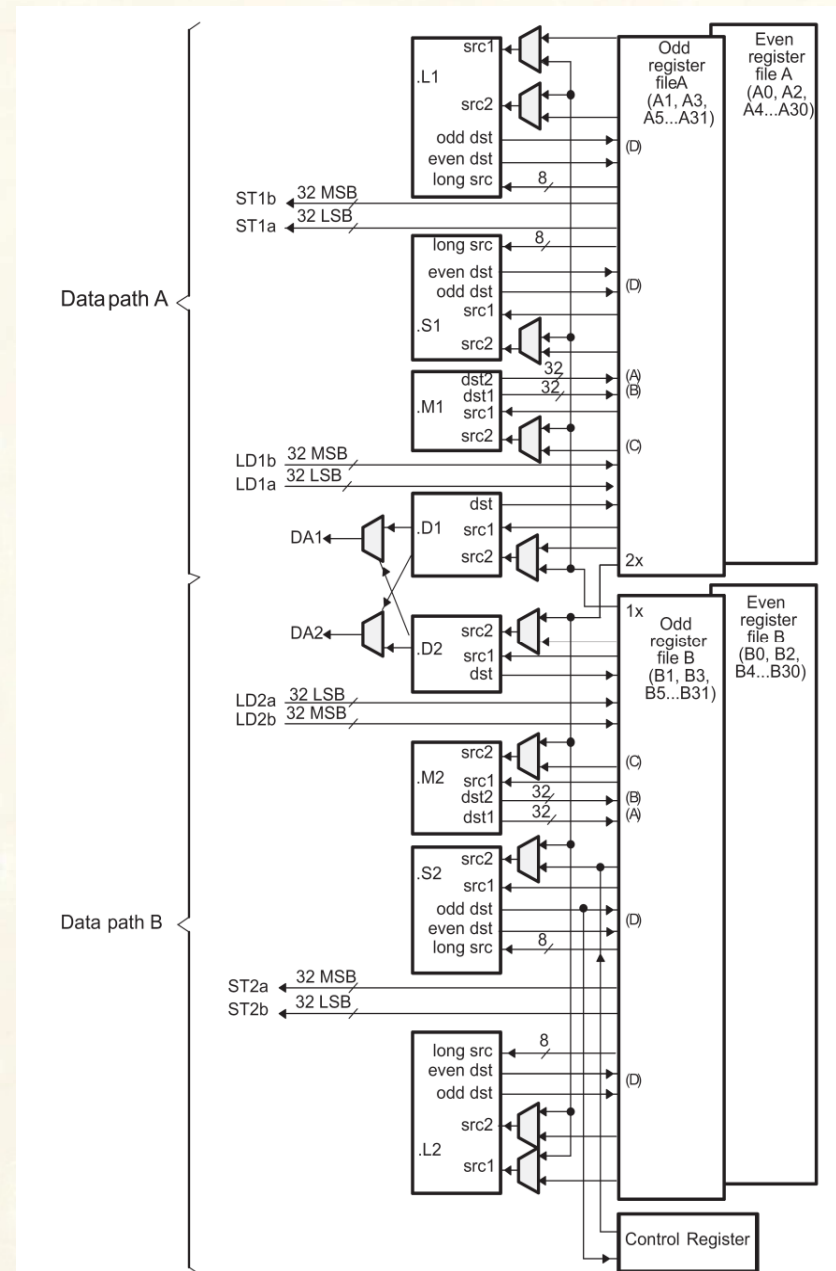
COEF:

.word B0,B1,B2,B3,B4, atd. ; Definice koeficientů X filtru

*Signálové procesory se
strukturou
VLIW*

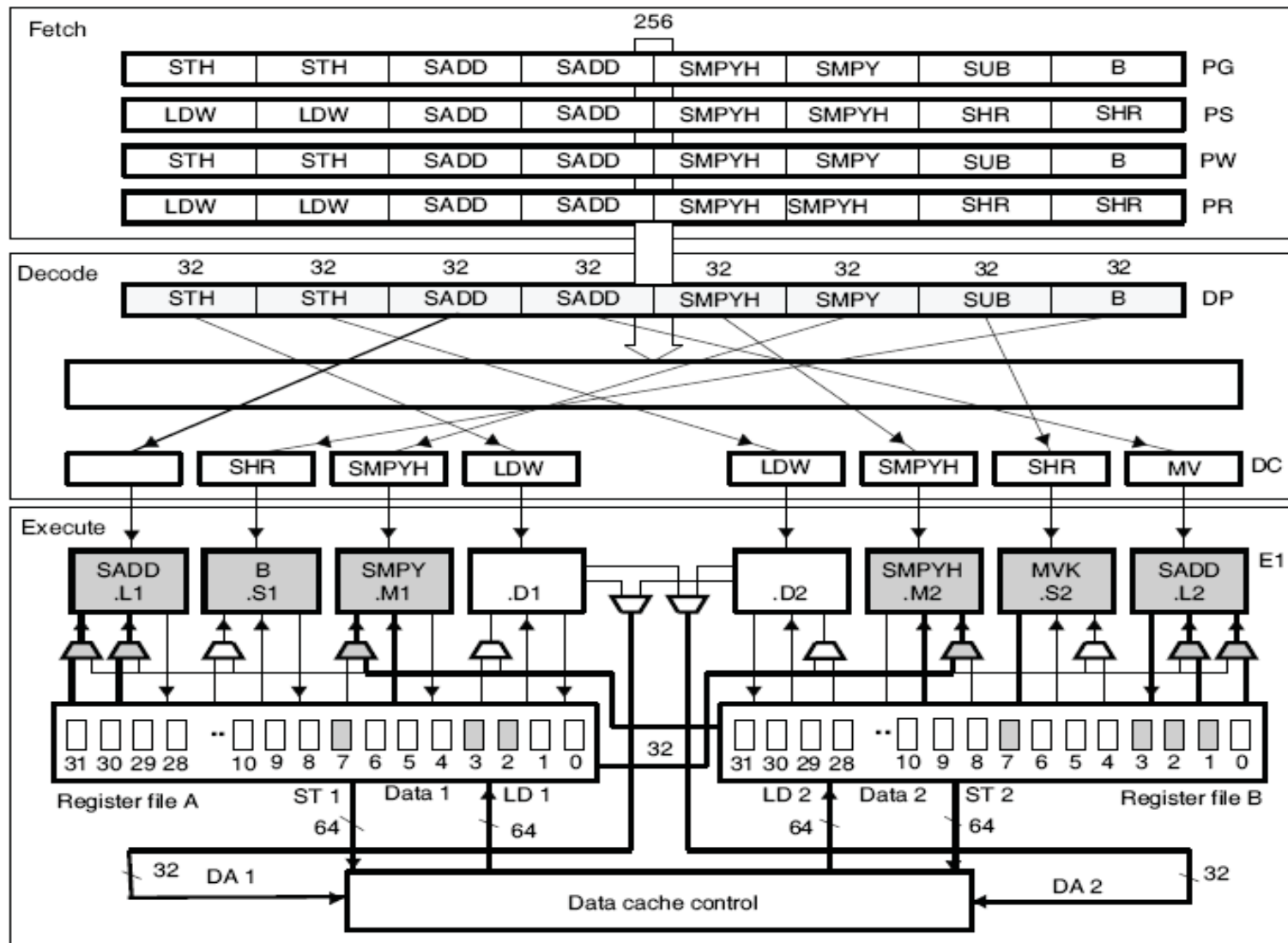
SIGNÁLOVÉ PROCESORY VLIW

- ❖ **Sdružená instrukce** u běžného signálového μP vyvolá současnou činnost MAC, ALU, Shifter, generátorů adres a AJ v oblasti čítače instrukcí.
- ❖ **VLIW μP** čtou 256 nebo 128 bitové instrukční slovo obsahující několik **instrukcí** určených pro jednotlivé aritmetické jednotky. Zpracování je paralelní nebo sekvenční.
- ❖ U TMS320C6xxx (TI, 1997) je to 8 32-bitových instrukcí určených pro dvě násobičky a šest dost podobných ALU, tím započal vývoj v oblasti
 - **Paralelní programování**
 - Programování s instrukcemi, jejichž výsledek je k dispozici až po několika strojových cyklech.



ZVYŠOVÁNÍ VÝPOČETNÍHO VÝKONU PROCESORŮ

- Procesory z řady TMS320C62xx a 64xx využívají 7 fázový pipeline a skrytě 11 fázový

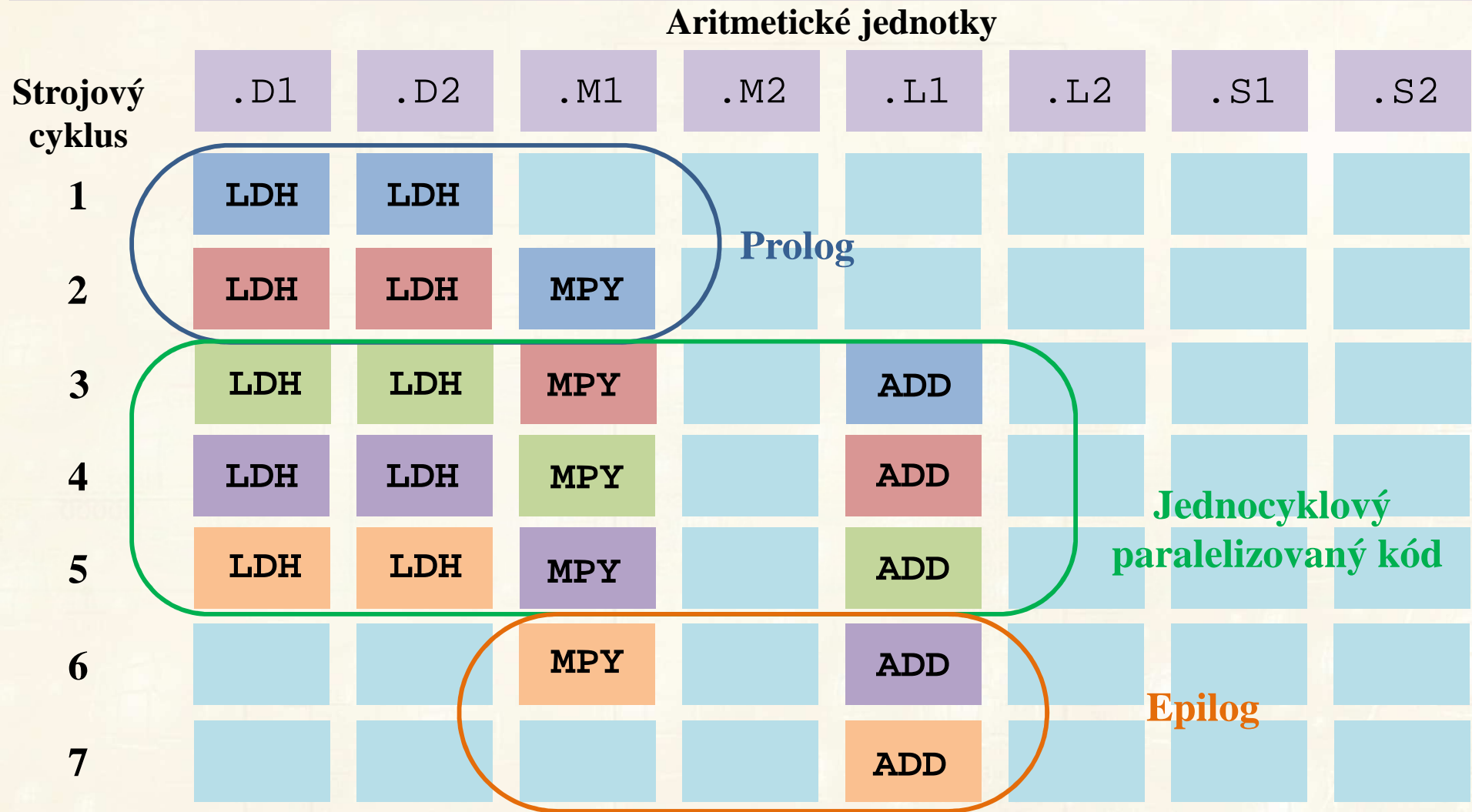


SEKVENČNÍ PROGRAM S ČÁSTEČNOU PARALELIZACÍ

Aritmetické jednotky

Strojový cyklus	.D1	.D2	.M1	.M2	.L1	.L2	.S1	.S2
1	LDH	LDH						
2			MPY					
3					ADD			
4	LDH	LDH						
5			MPY					
6					ADD			
7	LDH	LDH						
8			MPY					
9					ADD			

POSTUPNÁ PARALELIZACE PROGRAMU – PIPELINING CODE



Paralelizace (pipeline instrukcí) vyžaduje k vykonání 1/2 strojových cyklů!

PARALELNÍ PROGRAMOVÁNÍ NA PROCESORECH VLIW

Ukázka postupné paralelizace (tzv. Prolog) a výkonné fáze programu (filtru FIR) na procesoru se strukturou VLIW. Sériové řešení 640 cyklů, uvedené 28.

```
c0:      ldw .D1  *A4++,A5      c5_6:    ldw .D1  *A4++,A5
||      ldw .D2  *B4++,B5      ||      ldw .D2  *B4++,B5
||      || [B0] sub .S2  B0,1,B0
c1:      ldw .D1  *A4++,A5      || [B0] B .S1  loop
||      ldw .D2  *B4++,B5      ||      mpy .M1x  A5,B5,A6
|| [B0] sub .S2  B0,1,B0      ||      mpyh .M2x  A5,B5,B6

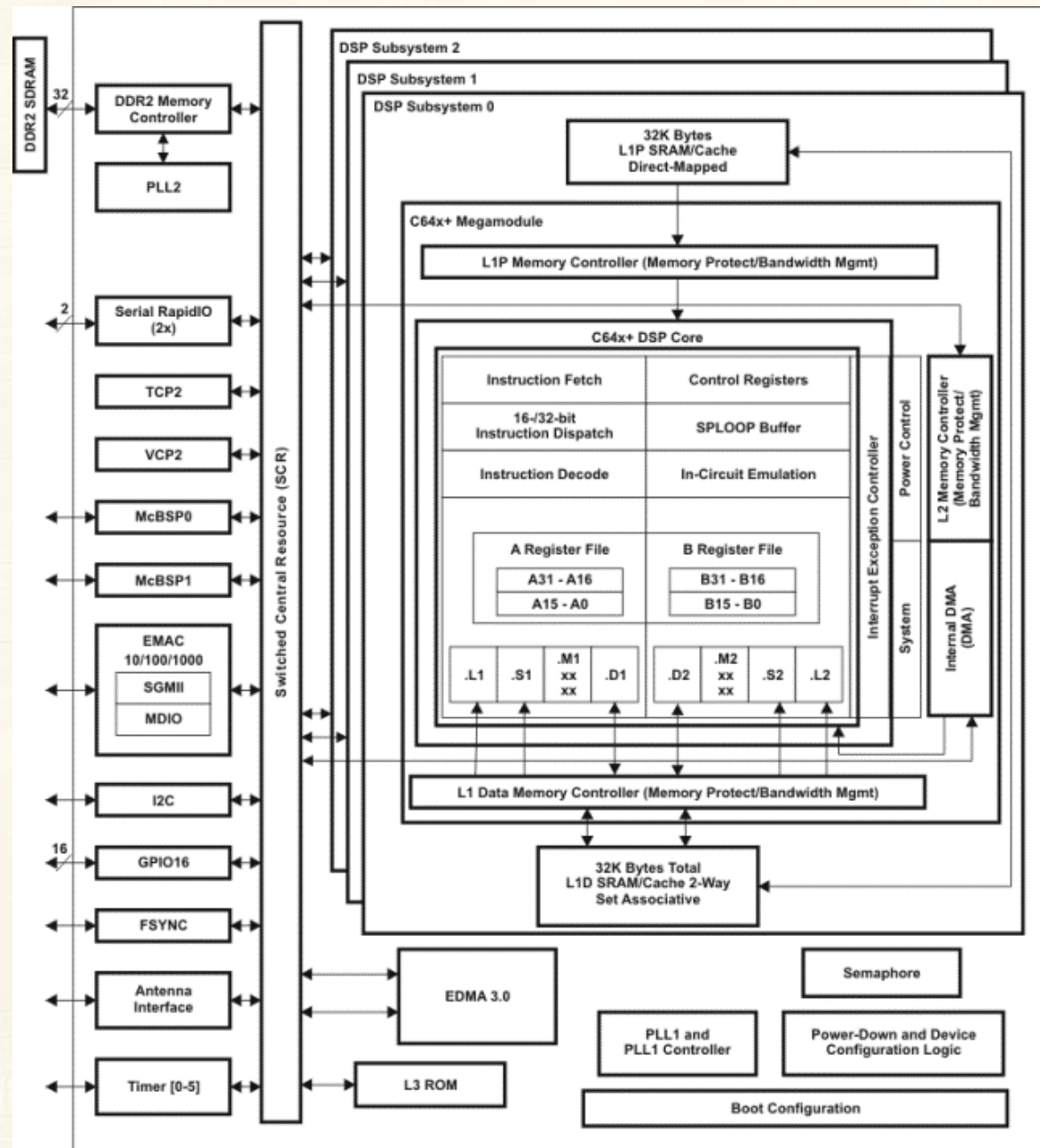
c2_3_4:  ldw .D1  *A4++,A5
||      ldw .D2  *B4++,B5
|| [B0] sub .S2  B0,1,B0
|| [B0] B .S1  loop
.
.
.
```

```
*** Single-Cycle Loop
loop:  ldw .D1  *A4++A5
||      ldw .D2  *B4++,B5
|| [B0] sub .S2  B0,1,B0
|| [B0] B .S1  loop
||      mpy .M1x  A5,B5,A6
||      mpyh .M2x  A5,B5,B6
||      add .L1  A7,A6,A7
||      add .L2  B7,B6,B7
```

VLIW SIGNÁLOVÉ MIKROPROCESORY S NĚKOLIKA JÁDRY

Donedávna nejvýkonnějším signálovým procesorem je procesor TMS 320C6474.

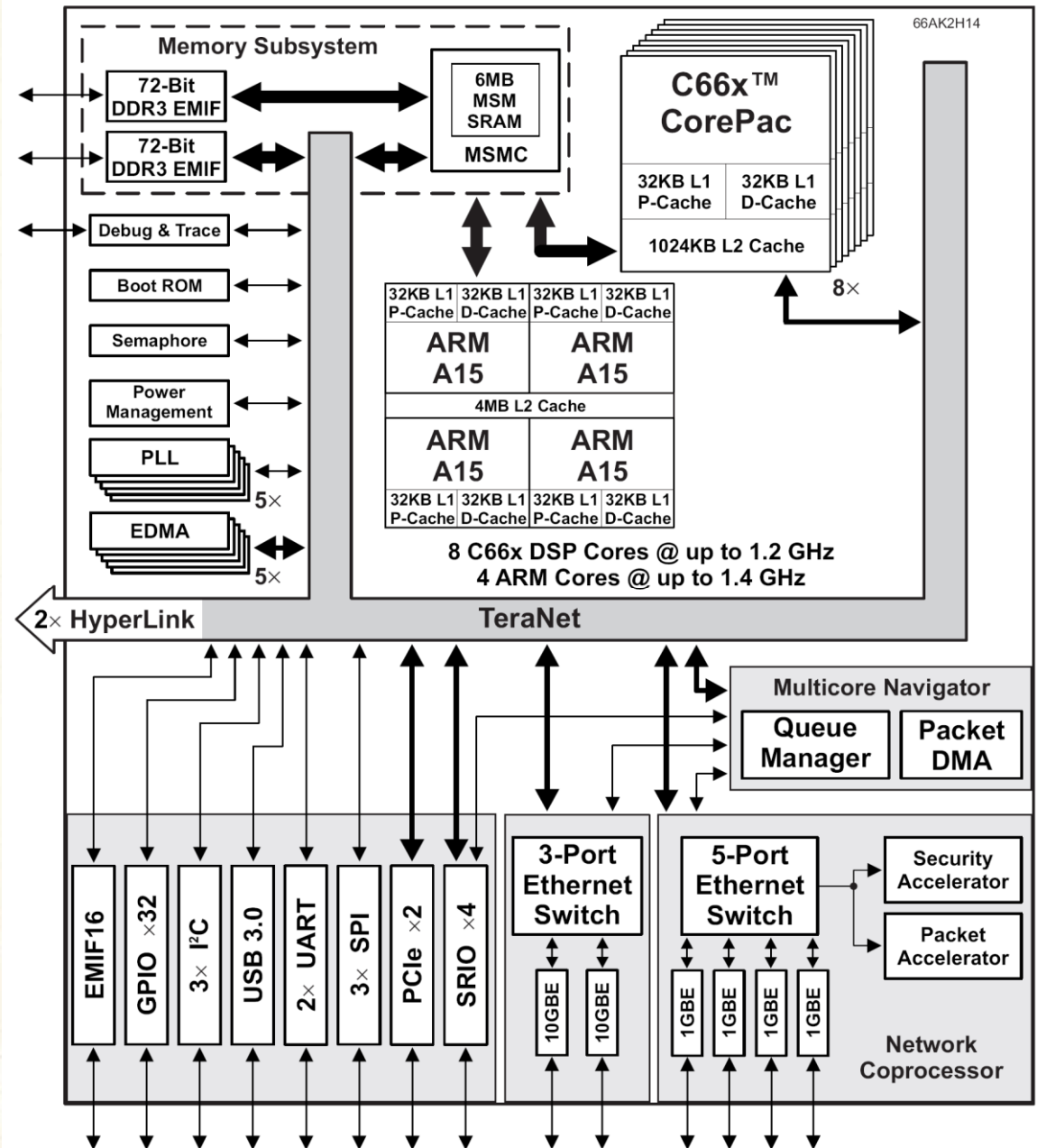
- Tři jádra VLIW řady 64xx
- Výkon 28400 MIPS/Core při 1,2 GHz
- Modifikované násobičky M1 a M2, které v jednom strojovém cyklu vypočtou jeden součin 32x32bitů, dva součiny 16x32, čtyři součiny 16x16 nebo 8x8 se sčítáním/odčítáním včetně jednoho komplexního násobení.
- Řada rozhraní jako I²C, 100Mb/s McBSP0, kontrolér Ethernet a rozhraním pro DDR2, atd.



VLIW SIGNÁLOVÉ MIKROPROCESORY S NĚKOLIKA JÁDRY + ARM

Nyní nejvýkonnějším signálovým procesorem je procesor TMS 320C66x plus 4xARM

- 8 jader VLIW řady 66x
- Výkon 38400 MIPS/Core při 1,2 GHz
- Modifikované násobičky M1 a M2, které v jednom strojovém cyklu vypočtou jeden součin 32x32bitů, dva součiny 16x32, čtyři součiny 16x16 nebo 8x8 se sčítáním/odčítáním včetně jednoho komplexního násobení.
- Doba instrukce 0,87ns



Copyright © 2016, Texas Instruments Incorporated

Vývoj programů

Použití daného programovacího jazyka závisí na

- ❖ Složitosti vyvíjené aplikace
- ❖ Architektuře a podpoře interních periférií vývojovým prostředím
- ❖ Na výkonu procesoru

Program pro mikropočítač můžeme vytvářet ve:

- ❖ **Strojovém kódu** – tj. v kódech instrukcí daného procesoru
- ❖ **Jazyce symbolických adres** – tj. v mnemonickém zápisu instrukcí daného procesoru
- ❖ **Jazyce symbolických adres s aritmetickou knihovnou**
- ❖ **Jazyce C**
- ❖ **Vyšších jazycích**
 - **Kompilované**
 - **Interpretované**

VÝVOJ PROGRAMŮ – MOŽNOSTI ZÁPISU PROGRAMU

The screenshot displays the µVision IDE interface for a project named "Blinky_4_Stopyk". The main window shows the C source code in "Blinky.c" and its corresponding assembly code in the "Disassembly" window. The registers window on the left shows the state of the processor registers, with the Program Counter (R15) at 0x080005BE. The GPIOA peripheral registers are visible on the right. The Command window shows the "Setup();" command, and the Memory window displays the memory dump starting at address 0x20000000.

Annotations in blue text with arrows point to the following elements:

- Strojový kód** (Machine code) points to the assembly instructions in the Disassembly window.
- Assembler - Memonické označení instrukce** (Assembler - Mnemonic instruction label) points to the assembly instructions.
- Jazyk C** (C language) points to the C source code in the Blinky.c window.

- ❖ **Strojový kód** – dnes se nepoužívá. Vložení instrukce vede k předadresování návěští a podprogramů.
 - Umožňuje plně využít možností instrukčního souboru.
 - Realizace je nesmírně zdlouhavá, možnost řady chyb.
 - Se strojovým kódem se setkáme hlavně při **podezření na chybu překladače** nebo jinak těžko vysvětlitelnou chybu.
- ❖ **Jazyk symbolických adres (JSA)** – zápis programu v **mnemonických označeních jednotlivých** instrukcí s odkazy na **symbolické adresy** (návěští), **konstanty**, **proměnné**, atd.
 - Používá se - **knihovní funkce jazyka C** a ovladače.
 - K části programu - **neefektivně realizovatelného** ve vyšším jazyce, **k obsluze nové periferie nepodporované vývojovým prostředím**.
 - V případech nezbytné **kontroly časového zpoždění**.
 - **Umožňuje nejefektivnější řešení dané úlohy v závislosti svých instrukcí, nápaditosti a znalostí programátora.**

- ❖ **Jazyk symbolických adres s aritmetickou knihovnou**
 - Využíval se pro složitější operace v době, kdy nebyly k dispozici překladače vyššího jazyka pro mikropočítače.
 - Knihovna obsahovala **podprogramy** s definovaným **vstupem a výstupem** operandů v jednotlivých registrech.
 - **Stejně jsou řešeny funkce a operace ve vyšších jazycích. Operandy jsou uloženy do registrů a následně je zavolán příslušný podprogram.**
- ❖ **Jazyk C** – je využíván k realizaci **složitých programů** pro jednočipové, signálové a částečně i PC procesory.
 - Zdrojový kód je **přenositelný** i na jiné typy procesorů. Pro každý procesor **musí být přeložen do strojového kódu.**
 - Vývoj je **podstatně rychlejší a bezpečnější**, než v JSA. Nelze v něm dosáhnout **efektivní realizace** některých operací podporovaných **instrukčním souborem procesoru. Neumožňuje** realizaci **riskantních operací** jako v JSA.

- Je-li napsán dobře, je délka strojového kódu větší, ale **srovnatelná** s velikostí kódu JSA.
- Obtížně realizovatelné operace v jazyce C nebo neefektivně kompilované mohou být do jazyka C vloženy ve formě assembleru (JSA). Pozor na případný **přenos programu na jiný procesor** (instrukční soubor, Big a Little endian).

Přenositelnost jazyka C

- Program bez odkazů na konkrétní registry nebo podprogramy v assembleru \Rightarrow bezproblémová kompilace na jiný procesor.
- Odkazy na registry nebo assembler musí být předělány pro nový typ procesoru. Po překladu zdrojového kódu bude strojový kód pro nový procesor funkční.
- Odlišná architektura nového procesoru může vést na **strojový kód nevyužívající jeho efektivnější instrukce**.
- Příklad: $y=x+x$ nebo $y=2*x$ nebo $y=x\ll 1$.
Překladač nemusí přeložit přesně Váš zápis.

Překladač jazyka C nekontroluje, zda byla nadefinovaná špatná velikost pole, odkaz na nedefinovaný prvek pole nebo na nevyužívané paměťové místo, atd. Příklad dvou zápisů

`unsigned char pole[15];`

`pole[20]=25;` - identifikován jako chybný

`alfa=26; pole[alfa-7]=36;` - není a nemůže být označen jako chybný

- Proto někdy bývá označován jako jazyk **nebezpečný**.

VÝVOJ PROGRAMŮ – MOŽNOSTI ZÁPISU PROGRAMU

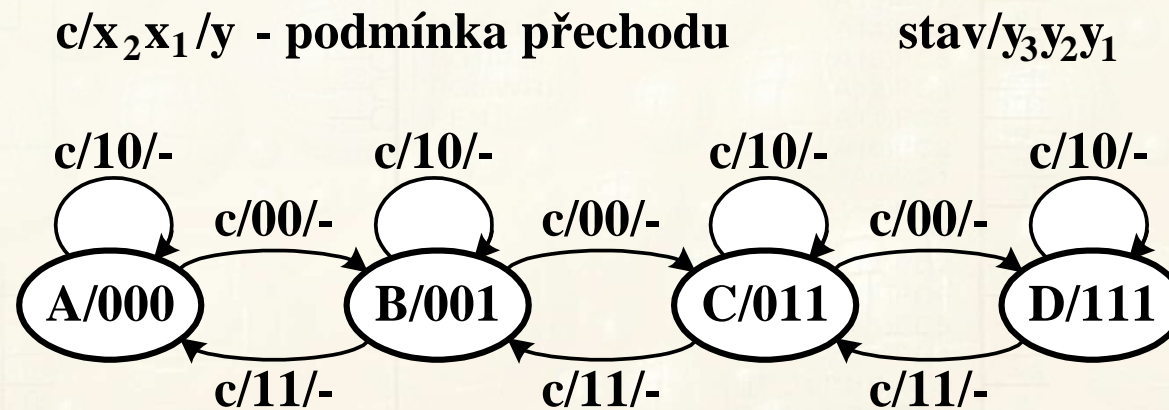
- ❖ **Vyšší a interpretované jazyky** – univerzálně nebo aplikačně orientované jazyky umožňující realizaci úlohy nebo problému.
- ❖ **Kompilované** – C, C++, Pascal, Basic, atd., ⇒ výsledkem je **strojový kód** naprogramovaný **do paměti s paralelním přístupem**. Zpracování je rychlejší, než jazyk interpretovaný.
- ❖ **Interpretované** – BASIC (zdrojový zápis), Java, PLC, (přeložené do **mezikódu (nikoliv strojového)**). Kompilovaný **aplikační program (interpret)** interpretuje **uživatelský program** na platformě daného procesoru. ⇒ Interpretovaný program je pomalejší a může být uložen v paměti RAM, EEPROM, sériové Flash nebo zálohované RAM. **Program nemusí** být uložen v paměti s paralelním přístupem ⇒ postupně čten a realizován interpretem.
 - Využití těchto jazyků má kořeny v multiplatformní přenositelnosti (Java)

- ❖ Dělení programů podle použití a způsobu zpracování
 - **Jednoprůchodové**
 - **Víceprůchodové**
 - **Rekurzivní** – nevhodný pro procesory s omezenou nebo malou kapacitou zásobníku.
- ❖ **Programy pro PC**
 - **Jednovláknové**
 - **Vícevláknové**
- ❖ Dělení podle reakce a rychlosti zpracování události
 - **Programování v reálném čase** – ČZS. **Jednočipové procesory** – jednotky kHz. **Signálové procesory** - desítky kHz. **VLIV** - až jednotky MHz. Vyšší pásmo – **běžné a DSP programovatelné obvody** – desítky a stovky MHz.
 - **Událostní programování**

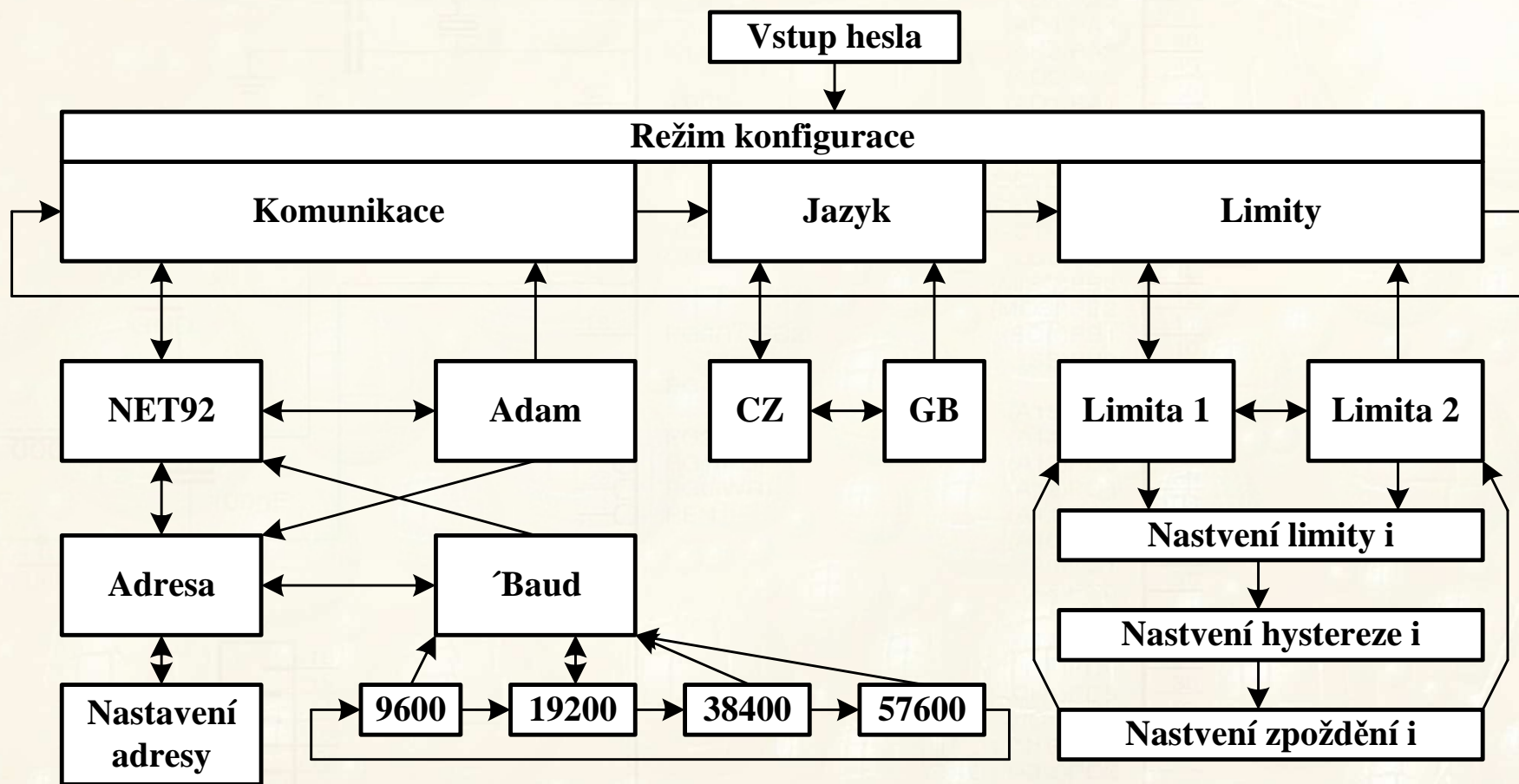
Rozdíl mezi událostním a programování v reálném čase?

❖ Dělení programů podle podobnosti

- **Stavové programování** – chování podobné **stavovému automatu**. Možné využít místo **RTOS** k realizaci dvou a více procesů, které běží současně (měření a řízení technologického procesu, měření a obsluha stromového konfiguračního menu, atd., příjem a zpracování komunikačních protokolů s proměnnou délkou).



STAVOVÉ PROGRAMOVÁNÍ – PŘÍKLAD KONFIGURAČNÍHO MENU



→ Přechod ovlivněný klávesnicí

STAVOVÉ PROGRAMOVÁNÍ - PŘÍKLAD

```
//Převod hodnoty Tx_data na posloupnost bitů sériového přenosu UART
If (Data_platna == true)
{
    vysli(0);          // vyslání start bitu
    for (i=0; i<8; i++)
    {
        if(Tx_data&0x01)!=0) vysli(1); else vysli(0);
        Tx_data=Tx_data>>1;}
    vysli(1); }      // vyslání stop bitu

//Realizace pomocí stavového řešení (řešení nejsou identická)
switch(Tx_State_var)
{
    CASE 0:          // čekání na nová data
    if (Data_platna == true) Tx_State_var = 1; break;
    CASE 1:          // vyslání start bitu, nulování počítadla
    vysli(0); Tx_State_var = 2; pocitadlo = 0; break;
    CASE 2:          // vysílání bitů 0-7
    if ((Tx_data & (1<<pocitadlo)) == 0) vysli(0);
    else vysli(1); pocitadlo++;
    if (pocitadlo == 8) Tx_State_var = 3; break;
    CASE 3:          // vyslání stopbitu
    vysli(1); Tx_State_var = 0
}
}
```

VÝVOJ PROGRAMŮ – MODULÁRNÍ PROGRAMOVÁNÍ

- ❖ Jednodušší a přehlednější vytváření rozsáhlých programů
- ❖ Nezbytné v případě **týmového vytváření programu**.
- ❖ Snazší ladění a úpravy podprogramů, možnost jejich násobného použití a snadného začlenění do programových knihoven.
- ❖ Důležité je správné definování vstupů a výstupů daného modulu (podprogramu), předávání parametrů mezi moduly.
- ❖ Odladěné a ověřené moduly lze spojit do výsledného i velmi rozsáhlého programu nebo začlenit do knihoven.
- ❖ Na stejném principu se vytváří programy v jazyce C, kde **proměnné jsou z datové paměti přeneseny do registrů** a následně jsou volány standardní knihovní moduly (podprogramy).
- ❖ Část parametrů se může předávat v **registrech** a zbývající parametry se přenáší přes **zásobník** nebo **paměťové umístění**.
- ❖ Při modulární tvorbě programu jsou větší nároky na **velikost zásobníku** ⇒ postupně je voláno **velké množství podprogramů**. Je-li zásobník součástí vnitřní paměti procesoru (omezená kapacita), může nás tento postup dostat do **potíží**.